Homework 1
CSE 5314
Rui Huang
2/18/2004

Exercise 1.8 Let *L* be a list of two elements *x* and *y*. Prove that there is an optimal offline algorithm OPT for *L* that satisfies the following properties: 1) OPT does not use paid exchanges; and 2) whenever there is a run of two or more consecutive requests for *x* (*y*), OPT moves *x* (*y*) to the front (if it is not already there) after the first request (of this run) using free exchanges.

Without losing generality, let the initial list be $L = xy$. The following observations can be made:

1) If the input sequence contains at least one *y*, then any algorithm will have a cost at least 1.
2) If the input sequence contains $(yx)^k$, the any algorithm will have a cost at least *k*.
3) If the input sequence contains $(yx)^k y$, the any algorithm will have a cost at least $k+1$.

From Table 1.1, for all three input sequence type, $x^i yy$, $x^i (yx)^k yy$ and $x^i (yx)^k x$, OPT has a cost of 1, $k+1$ and *k*, respectively. Comparing the above observations with the results from Table 1.1, one can see that OPT performs as well as any algorithms for the three input sequence types. Since the three input sequence types cover all possible input sequences, we can say that OPT is indeed optimal.

Exercise 2.5 Prove Lemmas 2.1 and 2.2.

Lemma 2.1 Suppose that x is initially in front of BIT's list. Then after serving the sequence yx, with probability ¾, item x is at the front.

Proof of Lemma 2.1: Case by case analysis:

$L = xy$	$x_0 y_0$	$x_0 y_1$	$x_1 y_0$	$x_1 y_1$
$\sigma = yx$	$x_1 y_1$	$x_1 y_0$	$y_1 x_0$	$x_0 y_0$

Lemma 2.2 Immediately after BIT serves the sequence yxy, with probability ¾, item y is at the front (independent of the initial order of x and y).

Proof of Lemma 2.2: Case by case analysis:

$L = xy$	$x_0 y_0$	$x_0 y_1$	$x_1 y_0$	$x_1 y_1$
$\sigma = yxy$	$x_1 y_0$	$y_1 x_1$	$y_0 x_0$	$y_1 x_0$

$L = yx$	$y_0 x_0$	$y_0 x_1$	$y_1 x_0$	$y_1 x_1$
$\sigma = yxy$	$x_1 y_0$	$y_0 x_1$	$y_1 x_1$	$y_1 x_0$

Exercise 3.1 Prove that any page replacement algorithm (online or offline) can be modified to be demand paging without increasing the overall cost of any request sequence.

Any paging policy can be described using the following model. Given a fast memory area of certain size, its initial content and a sequence of page requests $(r_{1,} r_2, ...)$ as the input, a paging policy produces an output as a sequence of events $(e_{1,} e_2, ...)$, where e_i can be either of the following two types:

Replace(a, b): replacing page *a* in the fast memory with page *b* in the slow memory.
Access(a): accessing (i.e., reading or writing) page *a* in the fast memory.

Based on the model, a policy is *on-demand* if and only if every *Replace(?, x)* is immediately followed by an *Access(x)*. The *cost* of a policy is the total number of *Replace* events in the output sequence. For every input sequence $(r_{1,} r_2, ...)$, its *Access* event should be in the same order in the output sequence, i.e., *Access(* r_1 *)* should proceed *Access(* r_2 *)*.

Let ALG_{any} be any paging policy, and $ALG_{ondemand}$ be its corresponding on-demand policy. Given the same initial memory configuration and the same input sequence to both ALG_{any} and $ALG_{ondemand}$, our proof completes when following objectives are met:

1) every *Replace(?, x)* is immediately followed by an *Access(x)* in the output sequence of $ALG_{ondemand}$;
2) the number of *Replace* events is no more in $ALG_{ondemand}$ than in ALG_{any};
3) $ALG_{ondemand}$ produces the same order of *Access* events as ALG_{any}.

To accomplishes the above, we provide the following mapping algorithm that takes the output sequence of ALG_{any} and generates the output sequence of $ALG_{ondemand}$. The mapping algorithm maintains the fast memory content, M_{any} ($M_{ondemand}$) of ALG_{any} ($ALG_{ondemand}$). For any page in $M_{ondemand}$, if the same page is also in M_{any}, then no link is provided. Otherwise, the page in $M_{ondemand}$ is linked to another page in M_{any}.

Initially, $M_{ondemand} := M_{any}$

For each event e_i the output sequence of ALG_{any}

When the event e_i = *Replace(a, b)*, do the following:

If $a \in M_{ondemand}$ then

If *a* is linked to another page, remove the link

If $b \in M_{ondemand}$, then

Move the link from *b* to *c* so that a is linked to *c* (1)

Else

Add a link from *a* to *b* (2)

Else

If $b \in M_{ondemand}$, then

Remove the link from *b* to *d*

Move the link from *c* to *a* so that *c* is linked to *d* (3)

Else

Move the link from *c* to *a* so that *c* is linked to *b* (4)

When the event e_i = *Access(a)*, do the following:

If there is a link from another page, say *b*, to *a*, then

Remove the link

Replace *b* with a in $M_{ondemand}$

Output *Replace'(b, a)*

Output *Access'(a)*

We now examine the above algorithm case by case:

Case 1: Both *a* and *b* are in $M_{ondemand}$. Since *a* and *b* are cannot be in M_{any} at the same time (because of the *Replace(a, b)* event ALG_{any}), there must be a link from *b* to another page, say *c*, in M_{any}. In this case, *a* is now linked to *c*.

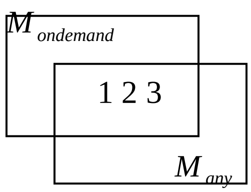
Case 2: Only *a* is in $M_{ondemand}$, but *b* is not. In this case, we just link *a* to *b*.

Case 3: *a* is not in $M_{ondemand}$, but *b* is. Since *a* is not in $M_{ondemand}$ but in M_{any}, there must be a page in $M_{ondemand}$, say *c*, that is linked to *a*. Similarly, since *b* is in $M_{ondemand}$ but not in M_{any}, then *b* must be linked to a page, say *d*, in M_{any}. In this case, *c* is now linked to *d*.

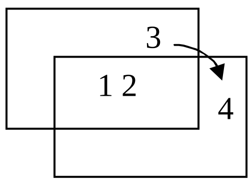
Case 4: Neither *a* or *b* is in $M_{ondemand}$. In this case, the original link from *c* is now linked to *b*.

Example: Let M_{any} = {1, 2, 3} initially, and let events in ALG_{any} be the following:

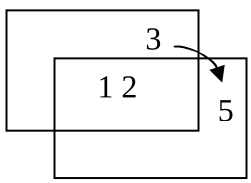
	ALG_{any}	$ALG_{ondemand}$



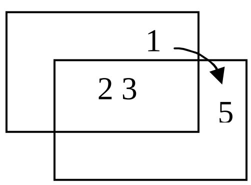
Replace(3,4), case (2)



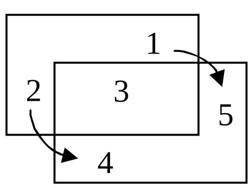
Replace(4,5), case (4)



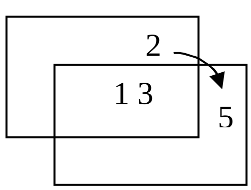
Replace(1,3), case (1)



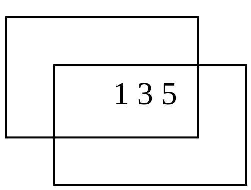
Replace(2,4), case (2)



Replace(4,1), case (3)



	Access(5)	Replace'(2,5) Access'(5)



The mapping algorithm accomplishes the three objectives we listed earlier:

1) Since the mapping algorithm outputs a *Replace'(b, a)* immediately followed by *Access'(a)*, the output sequence generated is valid for an on-demand policy.
2) For every *Replace* event the mapping algorithm encounters, it generates at most one new link. A *Replace'* event is generated only there is a link. Therefore, the total number of *Replace'* is not more than the total number of *Replace*. Thus, the cost of $ALG_{ondemand}$ is no more than that of ALG_{any}.
3) For every *Access(a)*, the algorithm generate an *Access'(a)*. Therefore, the order of the *Access* events are preserved.

Furthermore, since the mapping algorithm reads one event at a time, it is suitable to convert online policies.